The Abstract Accountability Language: its Syntax, Semantics and Tools

Walid Benghabrit, Hervé Grall, Jean-Claude Royer, Anderson Santana de Oliveira

The Abstract Accountability Language: its Syntax, Semantics and Tools

Walid Benghabrit · Hervé Grall · Jean-Claude Royer · Anderson Santana de Oliveira

2018-07-14

Abstract Accountability is the driving principle for several of regulatory frameworks such as the European Union's General Data Protection Regulation (EU GDPR), the Health Insurance Portability and Accountability Act (HIPAA) and the Corporate and Auditing Accountability and Responsibility Act, thus influencing how organizations run their business processes. It is a central concept for enabling trust and assurance in cloud computing and future internet-based services that may emerge. Nevertheless, accountability can have different interpretations according to the level abstraction. This leads to uncertainty concerning handling and responsibility for data in computer systems with outsourcing supply-chains, as in cloud computing. When defining policies to govern organizations, we need tools to model accountability in rich contexts, including concepts like multiple agents, obligations, remediation actions and temporal aspects. The Abstract Accountability Language (AAL) is built on logical foundations allowing to describe real-world scenarios involving accountability concerns. Its semantic principles provide us means to answer whether the conditions to reach accountability in a given context are met. Moreover, we created a tool support to verify and monitor accountability policies. The present paper recaps the main features of AAL, and introduces new contributions on expressiveness and tool support.

Keywords Accountability, Formal specification, First-order temporal logic, Semantics, Tool support, Verification

W. Benghabrit
Idento, 4 avenue Laurent Cély Asnières-sur-Seine, France - 92600 Ile-de-France
E-mail: walid.benghabrit@imt-atlantique.fr

H. Grall, J-C. Royer
Institut Mines-Télécom Atlantique, 4 rue A. Kastler, France - 44307 Nantes cedex 3
E-mail: Herve.Grall@imt-atlantique.fr,Jean-Claude.Royer@imt-atlantique.fr

A. S. de Oliveira
SAP Labs France, 805 avenue du Dr Donat Font de l'Orme, France - 06250, Mougins, Sophia Antipolis
E-mail: anderson.santana.de.oliveira@sap.com

1 Introduction

Demonstrating accountability is the main goal for organizations that need to comply with corporate policies and regulations. This involves defining control objectives and controls that span across organizational units, outsourced service providers, roles, and event technological infrastructures. The concept of accountability draws the central elements for many regulations, let us mention the European Union's General Data Protection Regulation (EU GDPR) [1], the Health Insurance Portability and Accountability Act (HIPAA) [2] and the Corporate and Auditing Accountability and Responsibility Act (also known as SOx) [3], to cite a few. Given its importance, it is paramount to understand under which conditions accountability can be met. In the digital era, the question involves controlling usage of information flows across organizational boundaries. Defining roles and accountability for the multiple actors handling data is not straightforward. The very first step is modelling organizational processes, their agents, and the interplay between the obligations they must execute in the processes, the authorizations, and the remediation actions in case of failures. Although the topic of accountability for distributed critical systems has gained importance over the years, there are few frameworks to help analyzing whether suitable properties apply to a given context today. Reviewing the related work in

formal accountability we observe a lack of concrete languages and tools to specify and experiment with accountability policies.

We are interested in concrete approaches for reasoning about accountability for a diverse set of applications (cloud computing, mobile, etc.) where models are defined in a formal language with precise semantics that can be verified by tools, such as theorem provers, SAT solvers and model checkers. We focus on accountability policies for computer based systems, where events can be unambiguously observed and asserted. We consider out of the scope of our work typically human moral dilemmas, or determining whether force majeure conditions were met that justify not fulfilling an obligation (mostly external to the system in question), or yet determining if an excuse is valid. These questions require human judgment, and very frequently some sort of human mediator. In this work we explore the spectrum of automatable, reproducible and enforceable accountability rules.

In previous work [4,5,6,7] we introduced the Abstract Accountability Language (AAL) which is a declarative policy specification language based in First-Order and linear Temporal Logic, featuring constructs supporting role definitions, delegations, obligations, and usage control rules. Besides setting its syntax and semantics, in these papers we showed how AAL is used to check compliance with regulations, how to find inconsistencies in policies using static analysis, and a practical verification approach supported by a state of the art automatic prover.

The current paper brings new contributions to some important aspects for accountability management: expressiveness and tool support. We provide an expressive language covering the three dimensions of [8]: information, justification and punishment. Information is achieved via formal declarative statements comprising various privacy features, temporal modalities, audit and rectification aspects. Justifications are realized thanks to our verification principles which allow checking for contract compliance and to query contracts using property verification. Punishment can be made explicit in the accountability contract and enforced thanks to a monitoring mechanism. Furthermore, we introduce policy templates, generic specifications useful in determining the expected behaviour for an abstract policy. This mechanism is very useful, since often accountability is challenged in outsourcing chains. The paper also overviews the AccLab tool support, our dedicated application allowing a component based description, the edition, verification and monitoring of accountability policies.

The remainder of the paper is structured as follows. Section 2 presents the core AAL grammar, usage examples to express accountability and the translational semantics. The next section is devoted to verification principles covering consistency, compliance, and property verification. Section 4 describes our AccLab tool support. We present the state of the art in computer science and accountability in Section 5 focusing on formal models and verification. The reader being up on accountability may skip this section and directly go to Section 6 which compares AAL and AccLab with related work and discuss some limits. A last section summarizes our findings and describes future perspectives.

2 Abstract Accountability Language

This section introduces our AAL language, its semantics and accountability expressiveness through various examples. In our approach, we consider that an accountability clause should express three things: a *usage*, an *audit* and a *rectification*. The usage expression describes access control, obligations, privacy concerns, usage controls, and more generally an expected behavior. The audit and rectification expressions are similar to usage expression but dedicated to auditing, punishment and remediation. The audit expression defines a specific audit event which triggers the auditing steps. The rectification expression denotes actions that are done in case of usage violations. For instance, to punish the guilty party and to compensate the victim agent. We follow [8,9] which argue that punishments and sanctions are parts of accountability.

2.1 AAL Grammar

We present in Listing 1 the core AAL syntax, more details and extensions can be found in [10]. An AAL program declares a set of types, agents, data and services which define the context. Usage expressions denote access control, privacy, or other actions and properties relevant to this context. The usage expression, `ActionExp` , is the most general construction but we also provide a dedicated accountability clause (the `CLAUSE` construction) and templates to assist users in defining accountability. The main AAL notion is an action or a service call represented as a message `sender.action[receiver](parameters)`. Authorizations are denoted by the `PERMIT` and `DENY` keywords prefixing an action or an expression. The language provides Boolean operators, first-order quantifiers and linear temporal operators. This is a simplified grammar and it has syntactic constraints about typechecking ensuring the consistency of the provided and

required sides of services. Another important constraint is that nesting permission is not allowed, we consider only one level of permission but it applies to any expression containing actions.

Listing 1 AAL core grammar

```
// A. AAL program  is a set of clauses and declarations
AALprogram  ::= {Declaration | Clause}*
// B. Declaration part
Declaration ::= TypeDec | ServiceDec | AgentDec | DataDec
TypeDec ::= TYPE type [EXTENDS(type*)]
ServiceDec ::= SERVICE Id TYPES(type*)
AgentDec ::= AGENT Id [TYPES(type*)]
               [REQUIRED(service*) PROVIDED(service*)]
DataDec ::= DATA  Id TYPES(type*)
               [REQUIRED(service*) PROVIDED(service*)]
// C. Clause part
Clause ::= CLAUSE Id(Usage [Audit Rectification])
Usage ::= ActionExp
Audit ::= AUDITING ActionExp
Rectification ::= IF_VIOLATED_THEN ActionExp
// D. Action expression (usage, audit and rectification
    expressions)
ActionExp ::= Action | {PERMIT | DENY} ActionExp
              | NOT ActionExp | Modality(ActionExp)
              | Condition | ActionExp1 BinaryOp ActionExp2
              | Quant ActionExp | Predicate
// E. Action expression components
Action ::= agent1.service'['agent2']'(Exp)
Exp ::= Variable | constant | Id.attribute | Predicate
Predicate ::= @Id(arg*)
Condition ::= [NOT] Exp | Exp1 {== | !=} Exp2
              | Condition1 {AND | OR} Condition2
Quant ::= {FORALL | EXISTS} Variable
Variable ::= Id : type
Modality ::= ALWAYS | NEVER | SOMETIME | NEXT
BinaryOp ::= AND | OR | => | UNTIL | UNLESS
Id, type, service, agent, arg, constant, attribute, true, false
     ::= literal
```

We present, in the rest of this section, examples demonstrating the AAL expressiveness and specially for accountability policies.

2.2 Usage Expression

An AAL program appears in Listing 2 which declares few types, agents, and services. Some predefined types are provided like `DataSubject`, `DataController`, `DataProcessor`, `data` useful for data privacy expressions. The usage expression states a permission and a prohibition related to some purposes for the `KardioMon` agent. It expresses that in any state for any `D:Data` if the owner is `Kim` and the purpose is research or statistics then `KardioMon` is allowed to process `D`. Else `KardioMon` is denied to process the data from `Kim`. This also defines a simple behavioral obligation for `KardioMon`: In any state each data processing should be followed by a notification to a data controller. We consider internal actions (like `process`) and binary actions (like `notify`) which represent a communication from a client (the required side) to a server (the provided side). The prefix `@` declares a predicate, and an action without receiver as in `KardioMon.process(D, purpose)` is an internal action.

Listing 2 A simple AAL program

```
AGENT Kim TYPES(DataSubject) REQUIRED() PROVIDED()
AGENT KardioMon TYPES(DataProcessor) REQUIRED(notify)
                                     PROVIDED(process)
AGENT auditor TYPES(DataController)  REQUIRED()
                                     PROVIDED(notify)
TYPE Data EXTENDS(data)
TYPE Purpose EXTENDS(data)
SERVICE process TYPES(Data Purpose)
SERVICE notify  TYPES(Data Purpose)
CLAUSE usageExample(
// Processing only if the purpose is research or statistics
ALWAYS FORALL D:Data P:Purpose (D.owner==Kim AND
   (@research(P) OR @statistics(P)))
      => PERMIT KardioMon.process(D, P)
ALWAYS FORALL D:Data P:Purpose (d.owner==Kim AND
   NOT (@research(P) OR @statistics(P)))
      => DENY KardioMon.process(D, P)
// Processing is  followed by a notification to an authority
ALWAYS FORALL D:Data P:Purpose (KardioMon.process(D, P)
       => SOMETIME KardioMon.notify[Auditor](D, P)) )
```

In [7] we show that AAL subsumes XACML, the de facto standard for security language, on several points, namely its pure logical semantics and expressiveness regarding privacy concerns and data transfer. The language supports a type system with Boolean operators, hierarchies for actions, users, resources and roles. AAL allows discrete linear time and duration. It enables complex dependencies between authorizations and obligations, thus making possible the writing of various protocols. In addition to covering many classic characteristics needed for security it provides new features for expressing accountability policies.

2.3 Accountability Expressiveness

As previously explained an accountability clause (`CLAUSE`) is compound from three expressions which are usage, audit and rectification (see Listing 3). Audit and rectification are optional expressions with the same syntax as usage expressions.

Listing 3 An accountability clause

```
CLAUSE AccPolicy (
//  usage expression
(FORALL D:Data (D.owner==Kim) => (PERMIT KardioMon.usage(D) AND
                    PERMIT KardioMon.send[MapOnWeb](D))))  AND
(FORALL D:Data ((D.owner==Kim) AND KardioMon.send[MapOnWeb](D)))
AUDITING   // auditing
   Kim.alert[Auditor]() => SOMETIME Auditor.audit()
IF_VIOLATED_THEN // rectification
  (SOMETIME (Auditor.punish[KardioMon](tenEuros)
             AND NEXT (KardioMon.give[Auditor](tenEuros)
                       AND Auditor.transfer[Kim](tenEuros))))
               )
```

The usage expression defines two permissions for the `KardioMon` agent and a simple send action. The audit is triggered as soon as the data subject alert the auditor. The rectification is a simple punishment with compensation to the data subject. The informal meaning of a clause is: *Always the usage is satisfied or if violated and if an audit occurs a rectification will be done.*

This meaning addresses security but also unanticipated behaviors violating the usage expression. This view is compliant with [11] and governs both preventive security and deterrence. It is simpler to write for end users and also it provides several intuitive properties we will see later.

Nevertheless, usage phrases are expressive and thanks to the uninterpreted (abstract) use of actions and predicates we are able to define many more controls (see Listing 4). Here, we write a more complex accountability example using only a unique usage expression with more precise descriptions of violations and rectifications. The usage defines an authorization to process and a prohibition to send data from `Kim`. If `Kim` does not give consent to processing then, in case `KardioMon` processes a data, he will be denied to process data forever. Finally, if the data processor sends a `Kim` data he is punished and `Kim` is compensated.

Listing 4 A usage expression with accountability

```
// some authorizations
(ALWAYS FORALL D:Data (D.owner==Kim) =>
       (PERMIT KardioMon.process(D) AND
       FORALL X:Subject DENY KardioMon.send[X](D))) AND
// illegal processing and rectifying
(FORALL D:Data
   ((NOT Kim.giveConsent[KardioMon]("Agreed")) UNTIL
        ((KardioMon.process(D) AND (D.owner==Kim) AND
  NEXT (ALWAYS (FORALL D:Data DENY KardioMon.process(D)))))))
     AND
// illegal sending then punishing and compensating
(ALWAYS ((SOMETIME EXISTS D:Data X:Subject
   (KardioMon.send[X](D) AND (D.owner==Kim) AND (X!=Kim)))
              => (Auditor.punish[KardioMon]()  AND
                  Auditor.compensate[Kim](tenEuros))))
```

Following this example many variations can be considered for the audit time, for the data to be audited and the auditing mechanism. A punishment example is described in Listing 5, where `Kim.do()` is a denied action. The punishment has three cumulative steps, the two first oblige `Kim` to pay in case of misconduct. The last step prohibits login for `Kim` in case of a third violation.

Listing 5 Expression with three rectification steps

```
// first penalty
(ALWAYS (Kim.do() => Kim.pay[Sys](ten))) AND
// second step
(ALWAYS ((Kim.do() AND NEXT SOMETIME Kim.do())
                    => Kim.pay[Sys](fifty))) AND
// final sanction
(ALWAYS ((Kim.do() AND NEXT SOMETIME Kim.do() AND
                  NEXT SOMETIME Kim.do()) =>
               NOT Kim.login() AND ALWAYS DENY Kim.login()))
```

In Listing 6 we have another example denoting permissions, logging in case of modify and send actions, audit, punishment and compensation. The first part of the example shows a *generalized authorization*. `Alice` has the permission to modify and send any data, however she is denied to modify a data and then send it to `Bob`. Such a prohibition is an information which can be checked for consistency but also it can be violated and we should detect and rectify it.

Listing 6 An example with explicit log

```
// [1] strict permissions
ALWAYS FORALL Z:Agent D:Data
   ((D.owner==Kim) => (@relatives(Kim, Alice)
       <=> PERMIT Alice.modify(D) AND PERMIT Alice.send[Z](D)))
// [2] but prohibition to chain modify and send to Bob
AND DENY EXISTS D:Data SOMETIME (Alice.modify(D)
                        AND SOMETIME Alice.send[Bob](D))
// ...
// logging some actions by an external observer
AND FORALL D:Data ALWAYS (Alice.modify(D)
                          => Auditor.log("modify", Alice, D))
AND FORALL D:Data ALWAYS (Alice.send[Bob](D)
                          => Auditor.log("send", Alice, Bob, D))
// auditing and rectifying
AND FORALL D:Data ALWAYS
     ((Auditor.log("modify", Alice, D)
       AND SOMETIME Auditor.log("send", Alice, Bob, D))
    =>  (Auditor.punish[Alice](twenty)
       AND (EXISTS X:Agent ((d.owner==X)
              AND Auditor.compensate[X](ten))))
```

The second part of the example illustrates an explicit logging and audit in case of the violation. An external observer is assumed to log these critical actions and to detect the violations.

Listing 7 makes explicit the violation expression, `EXISTS S:Data Bob.send[Alice](S)`, which is assumed to be disjoint from the usage expression. It also extends the accountability clause in specifying a link between a judgment identifying culprits and victims, and the rectification. The behavior of this example is: always the usage is satisfied or else it is satisfied until a violation occurs and in case of audit there is punishment of culprits and compensation of victims.

Listing 7 Extended accountability example

```
// audit expression
ALWAYS (Kim.notify[Auditor]() => Auditor.audit()) AND
// judgment
ALWAYS FORALL D:Data (Bob.send[Alice](D) =>
            (@guilty(Bob) UNTIL Auditor.enforced[Bob]())
             AND (@victim(Kim) UNTIL Auditor.enforced[Kim]())
// behavior
AND (ALWAYS FORALL D:Data (PERMIT Kim.send[Bob](D)
                           AND DENY Bob.send[Alice](D))
    OR
    ((FORALL D:Data (PERMIT Kim.send[Bob](D)
                AND DENY Bob.send[Alice](D)))
    // until a violation occurs
    UNTIL ((EXISTS S:Data Bob.send[Alice](S)) AND
             (ALWAYS (Auditor.audit() =>
      ((FORALL A:Agent (@guilty(A) => Auditor.punish[A]())) AND
      (FORALL A:Agent (@victim(A) => Auditor.compensate[A]()))
        )))))
```

We limit the complexity of the sub-expressions for readability, which is a critical problem. To mitigate it we propose to use templates in Section 2.6.

2.4 Semantics

In this subsection we sketch the translation process from AAL to $\mathbb{FOTL}$, giving semantics to it. Additional

details can be found in [6]. Here, we concentrate on the main principles and two extensions: *general authorizations* and *templates*.

There are two cases for the translation:

1. The first case is to translate each usage expression in FOTL, and this applies to a clause without its optional audit and rectification expressions.
2. The second case is relevant for CLAUSE with its two optional sub-expressions. In this case we built a new temporal sentence with the three expressions, then an interpretation of this sentence is made. More generally, this is the process used by the template construction.

2.4.1 First-Order Temporal Logic

FOTL extends both first-order logic (allowing predicates and quantifiers) and propositional linear temporal logic (allowing temporal modalities). FOTL formulas are built from variables V, constants C and predicates with a fixed arity P_i. If P_i is a predicate symbol and (e_j) are variables or constants then $P_i(e_j)$ is an atom (A). In the grammar of Listing 8 one can recognize Boolean operators (`not`, `and`, `or`, `=>`), temporal operators (`next`, `until`, `always`, `sometime`) and quantifiers (`forall`, `exists`). Note that we use lower case letters for these operators to make more clear the distinction with AAL.

Listing 8 FOTL grammar

```
F ::= C + A + (not F) +  (F and F) + (F or F) + (F => F)
          + (forall V F) + (exists V F)
          + (always F) + (sometime F) + (next F) + (F until F)
```

The semantics of FOTL is based on models which are infinite sequences, indexed by naturals, of first order structures (a non empty domain and an interpretation of the atomic predicates). Such a model will be called a *trace* as usual in linear temporal logic.

2.4.2 Expression Translation

The function `[_]: AAL → FOTL` represents the translation from an AAL expression to a FOTL one. Types, predicates, actions, attributes are all translated into predicates. Boolean operators are convenient to express logical relations like type union or disjunction. Agents and data instances are defined as constants. The translation of these declarations leads to a logical context prefixed with an `always` modality. Interpretation of Boolean, quantifiers and temporal operators are straightforward in FOTL. The PERMIT and DENY modalities generate new opposite events and a new context property which states that processing an action implies it is permitted. The extended grammar supports few additional facilities to express time in actions, limited equality and a past temporal operators. We also provide translations for these features. This translation needs some care in managing existential and universal quantifiers, and in adding the logical sentences capturing equality, dates, and authorizations.

2.4.3 Accountability Clause Interpretation

The accountability clause with its three expressions is an ease to write accountability and its translation is more elaborated. Our accountability clause is defined in Listing 9, AE the audit expression is assumed to generate atomic `audit` events. The informal clause meaning is as follows: *audit is running and at each instant either usage is satisfied or it is violated and then if an* `audit` *event occurs there will be rectification.*

Listing 9 Accountability clause interpretation

```
[ALWAYS AE AND
 ALWAYS (UE OR ((NOT UE) AND (ALWAYS  audit => RE)))]
```

Listing 9 provides our default interpretation for accountability. As soon as a violation of the usage is done and if an audit event occurs later then rectification applies. Figure 1 illustrates the linear traces with a violation and rectification. Let UE a usage expression, a *violation* can be formally defined as an expression verifying `(VE => NOT UE)`. Note that there is no hypothesis on RE and it could allow one rectification only, one each time, only one in the future or stands forever in the future. However, this clause is restricted because we dot not make explicit logging or auditing actions and parameter passing between the three parts is limited to constants.

2.5 General Authorizations

We define a general means to describe complex permissions, not only covering the case of atomic expressions as in many previous work. There are many cases, in protocols, for instance where we want to specify some permissions for individual actions but in the same time to prohibit few chaining of these actions. For instance, in our Listing 6 we permit modify and send actions, but we do not want to chain a data modification and a send action. In our interpretation we have a sub-world for the authorizations and another one for the "real" actions. The distinction is made by generating new events associated to each permitted sequences of actions. The connection between both worlds is done by generating context rules like `always (expression => permit(expression))`. The function *permit* copies the expression and uniquely

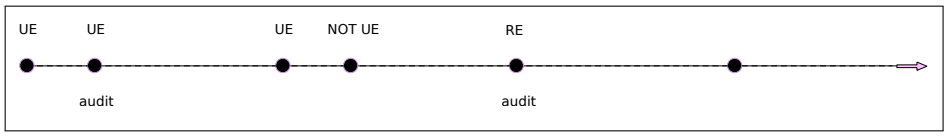

Fig. 1 A trace with a violation and rectification

tagged each actions, thus putting it in the world of permissions. We extend our initial semantics in the following manner: *i)* `DENY` is rewritten as `NOT PERMIT`, *ii)* for the construction `PERMIT expression`, without nested authorization, the expression is copied thanks to the function *permit*, *iii)* we add the clauses linking an expression with its permission, and *iv)* we enrich the context with rules to link permissions for complex expressions with action permissions. The translation of expressions `[1]` and `[2]` of Listing 6 appears in Listing 10 (but without the data type translations).

Listing 10 FOTL translation part of Listing 6

```
// from expression [1]
always forall X, D (modify(X, D) => P1_modify(X, D)) and
always forall X, Y, D (send(X, Y, D) => P2_send(X, Y, D)) and
always forall Z, D ((relatives(Kim, Alice) and owner(Kim, D))
      <=> (P1_modify(Alice, D) and P2_send(Alice, Z, D))) and
// from expression [2]
not exists D sometime (P3_modify(Alice, D) and
                       sometime P3_send(Alice, Bob, D)) and
// link between expression and permission
always ((exists D sometime (modify(Alice, D) and
            sometime send(Alice, Bob, D)))
        => (exists D sometime (P3_modify(Alice, D) and
                  sometime P3_send(Alice, Bob, D)))) and
// link between complex and atomic permissions
always ((exists D sometime (P3_modify(Alice, D) and
            sometime P3_send(Alice, Bob, D)))
        => (exists D sometime (P1_modify(Alice, D) and
                  sometime P2_send(Alice, Bob, D))))
```

This example captures that chaining modify and send requires the permission to do it but there is a conflict with the negative permission. However it does not prohibit to do the modify or send actions as far it concerns distinct data. Another useful feature for the user is the ability to set different interpretation modes. For instance, the `MODE: STRICT` generates `(always (action <=> permit (action)))` for the case at hand, it provides a context where, in addition to the default mode, if an action is permitted its negation cannot be performed. We also add `MODE: =>` which allows to relate permissions for complex usage expressions with action permissions as in the above example. The default mode corresponds to `<=>` denoting that a permission for a complex expression is equivalent to the expression of its atomic permissions. One remaining point is to find an efficient implementation of the general authorization feature. We are investigating a solution based on a unique normal forms for FOTL expressions.

2.6 Accountability Templates

We go further in assisting privacy and security officers in writing policies with the help of dedicated templates. A *template*, or a behavioral pattern, is a function which transforms several AAL expressions into a more complex AAL sentence expressing a specific security practice. The interpretation uses the translation function `[ ]` and applies it to the template instantiation with some parameters. It is quite straightforward to write few templates for authorizations, data retention, data transfer, transfer of ownership or permission. More complex templates concern accountability. For instance, the template associated to the previous accountability clause is defined as in Listing 11.

Listing 11 Accountability clause template

```
clause(AUDIT, AE, UE, RE) = ALWAYS AE AND
        ALWAYS (UE OR ((NOT UE) AND (ALWAYS  AUDIT => RE)))
```

Templates abstracting two punishment variations appear in Listing 12. The first template enables exactly one punishment in case of a violation (`VE`). It triggers the i^{th} punishment in case of i successive violations. While the second one generalizes the Listings 5 and accumulates the punishments for every sequence of violations. That is, if an execution trace contains i ($i \leq n$) successive violations then punishment actions are performed from 1 to i.

Listing 12 Punishment templates

```
// VE: violation expression
// P1 ... Pn : punishment expressions
// END: last deactivate the possibility of violation
// first exclusive punishment template
punish1(VE, P1, ... Pn, END) = (ALWAYS (END => NOT VE)) =>
((ALWAYS NOT VE) OR
(SOMETIME (VE AND NEXT ((ALWAYS NOT VE) AND (SOMETIME P1))))
OR ... OR
(SOMETIME (VE AND ... AND
              NEXT ((ALWAYS NOT VE) AND (SOMETIME END)) ... ))
// second cumulative punishment template
// same hypotheses as above
punish2(VE, P1, ... Pn, END) = (ALWAYS (END => NOT VE)) =>
(ALWAYS (VE => SOMETIME P1)) AND
...
(ALWAYS ((VE AND ... NEXT SOMETIME VE) => SOMETIME END))
```

Listing 13 describes a template for an extension of our accountability clause. Our example of Listing 7 is an instantiation of the template in Listing 13. This template makes explicit several new parameters: the `VE` violation expression and the `JE` the judgment expression,

and two conditions. The first condition states that UE and VE are disjoint. The second condition links the judgment expression JE with the violation VE and should identify culprits or victims. The qualification with predicates @guilty and @victim remains until enforcement by the agent A. The parameter instantiation for Listing 7 appears in Listing 14.

Listing 13 Extended template

```
extended(AUDIT, AE, UE, VE, JE, RE) =
// application conditions
(ALWAYS (VE => NOT UE)) AND
(ALWAYS ((JE AND VE) =>
     ((EXISTS A:Agent (@guilty(A) UNTIL Auditor.enforced(A)))
     OR (EXISTS A:Agent (@victim(A) UNTIL Auditor.enforced(A))))
     ))
 => // template
   (ALWAYS (JE AND AE) AND
((ALWAYS UE) OR (UE UNTIL (VE AND (ALWAYS (AUDIT => RE))))))
```

That means, replacing the template parameters (Listing 13) with these values (Listing 14) we should get the original accountability expression (Listing 7) or an equivalent expression.

Listing 14 Parameters instantiation

```
AUDIT = Auditor.audit()
AE = (Kim.notify[Auditor]() => AUDIT)
UE = FORALL D:Data (PERMIT Kim.send[Bob](D)
                    AND DENY Bob.send[Alice](D))
VE = EXISTS S:Data Bob.send[Alice](S)
JE = (VE => ((@guilty(Bob) UNTIL Auditor.enforced[Bob]()) AND
            (@victim(Kim) UNTIL Auditor.enforced[Kim]())))
RE = (FORALL A:Agent (@guilty(A) => Auditor.punish[A]()))
    AND (FORALL A:Agent (@victim(A) => Auditor.compensate[A]()))
```

Dependencies between formal parameters are permitted if they are not circular. Furthermore, we will show in Section 3.3 that this instantiation can be automatically verified. As the previous accountability clause, this template enjoys few properties described in Listing 17 and 18.

We expect to enrich this initial set of templates, the challenges are: *i)* to get a catalogue of accountability practices with a precise while informal description and *ii)* to formalize them in AAL and to prove their expected properties.

3 Verification and Tool Principles

The goal of this section is to introduce the principles behind the AccLab tool support. We focus here on software engineering activities related to policy verification. The objectives of verification are various, comprising: To prove some expected properties, to detect undesirable situations, or to eliminate redundancies. A redundancy is defined, in our context, as a policy P for a policy set R, satisfying R => P is valid. It can be checked using the prover back-end, but it is not a critical problem since it does not entail the system logic. Furthermore, we claim that it is practically uncommon in complex accountability policies with linear time and quantifiers. Thus we consider that the conflict detection question is really more critical, and formal policy compliance is an important requirement in regulations. From the related papers we can classify the properties of interest as internal properties or specific properties. Internal properties are related to the system and independent from the business domain like consistency, completeness, simplification, conciseness, and so on. Specific properties are depending on the business domain, several classes are relevant in our context. We consider specific properties related to privacy or security concerns but also some proper accountability properties.

To perform verification we rely on the temporal resolution procedure presented in [12]. This procedure has been implemented in a theorem prover (TSPASS [13]), which we have integrated into our specification and verification environment, the AccLab tool. The decision procedure behind the prover addresses the so-called *monodic* fragment [14,15]. The monodic condition states that any temporal sub-formula has, at most, one free variable. This is a constraint which is satisfied in our examples. It allows to mix, in a non trivial way, linear temporal logic and first-order logic, thus providing expressiveness. Regarding efficiency, the temporal resolution behind TSPASS has a non elementary complexity. Nevertheless, there are three papers which demonstrate the ability of this tool to solve real examples [16, 12,17]. The latter shows in fact that TSPASS is competitive with model-checkers and SAT solvers. In addition [18,6] study several non trivial examples with temporal operators and unbounded data with acceptable performances. In the rest we focus on three activities: conflict management, checking compliance and proving accountability properties.

3.1 Conflict Management

Conflict management consists of three activities: detection, localization (that is, finding the conflicting rules, or more generally finding the unsatisfiable core in the policy set) and conflict resolution, this is a key issue. There are various techniques, the most common principle is to check for pairs of incompatible rules. But this is not generally a correct approach since it can forget some conflicts. A more general way is to look for conflicts between any number of rules. It is still weak, inefficient and enforces a too strict writing style. In a context where we do not have rule, like with AAL, the correct approach is to rely on satisfiability or logical consistency and to use a solver, prover or model-checker. This is the approach we use with AAL, the

TSPASS prover allows us to check for the satisfiability of a set of policies. However, the drawback is that localization becomes less obvious than looking for pair of rules. But we reused the masking principle suggested in [19], which is intuitive and simple to implement. We consider that resolution is a manual activity in charge of the specifier.

Writing inconsistent specifications is rather common, one example is our Listing 4 where we can find a problem as soon as we add the fact that `Kim` owns some private data. Indeed there is a conflict between the permission and the prohibition to process by `KardioMon`. Listing 15 shows the result provided by our tool and the solution is to remove the permission for `KardioMon` to always process.

Listing 15 Conflicting example

```
// The example is unsatisfiable, there is a conflict
// with the following expressions
PERMIT KardioMon.process[KardioMon](D)
EXISTS private:Data (private.owner == Kim)
NEXT ALWAYS FORALL D:Data (DENY KardioMon.process[KardioMon](D))
```

3.2 Policy Compliance

In accountability contracts, compliance covers different meanings. A common idea, as in [20,21], is to check if a given real situation or execution trace is compliant or according to the contract. This is often the basis of the audit and detection mechanisms. We called it *testing compliance* and while this kind of compliance can be tested with our tool we are rather concerned with *contract compliance*. The goal of the contract compliance is to ensure that the provider's contractual terms satisfy or ensure the client's requirements. Thus it is much more complex than testing compliance since it is about proving that a policy is "stronger" than another one. In propositional logic it could be as simple as the implication of `Provided => Required`. In the FOTL framework the semantics is linear trace based and the previous intuition is also correct in the sense that the traces satisfying the provider clauses are included in those satisfying the client expectations.

Listing 16 A provider compliant policy

```
CLAUSE ListingProvider (
(FORALL D:Data S,K,M,A:Agent ((D.owner==S)
        AND (PERMIT K.usage(D)) AND PERMIT K.send[M](D)
        AND K.send[M](D) AND K.notify[A]()))
AUDITING (FORALL P,A:Agent
         (P.notify[A]() OR P.alert[A]()) => A.audit())
IF_VIOLATED_THEN (FORALL M:Money
   (auditor.punish[KardioMon](M) AND
       (NEXT (KardioMon.give[auditor](M)))
              AND auditor.transfer[Kim](M)))
)
```

More sophisticated semantics exist, but this one provides an intuitive and simple interpretation which is suitable for end-users. Thus compliance between two expressions relies on the validity of the logical implication, that is the validity of `R => (Provided => Required)`. For instance, Listing 16 presents a provider policy which ensures the clause in Listing 3. While this example seems rather simple, a tool is convenient in establishing the compliance, here in less than one second.

The `CLAUSE` construction of Listing 11 enjoys an interesting property: The contract compliance between two accountability clauses can be established with more natural (monotonic/covariant) conditions (see Listing 17).

Listing 17 Natural Condition for Accountability Clauses Compliance

$$((\texttt{ALWAYS}\ (UE_p \Rightarrow UE_r)) \ \texttt{AND}\ \texttt{ALWAYS}\ (AE_p \Rightarrow AE_r) \ \texttt{AND}\ (\texttt{ALWAYS}\ (RE_r \Rightarrow RE_r))) \Rightarrow \texttt{CLAUSE}(\texttt{AUDIT}, AE_p, UE_p, RE_p) \Rightarrow \texttt{CLAUSE}(\texttt{AUDIT}, AE_r, UE_r, RE_r)$$

Where `AE`, `UE`, `RE`, are respectively, the audit, usage, and rectification expressions and $_p$ (respectively $_r$) is the server (or provided) side (respectively the client/required side). We also demonstrate that this provides a more efficient way to check for compliance by splitting the formula in three shorter verification parts. For instance, in [6] we show an example of accountability compliance, with more than 1200 identifiers, which does not finish before 10000s with the global compliance formula. Using the above criterion it succeeds in less than 1400s. The use of additional heuristics can reduce the proving time down to 4s.

3.3 Specific Property Verification

Verification of properties needs the manual translation of the property into AAL and then the use of the prover. As previously explained the principle is to check the validity of `R => Property`. The prover, with the monodic restriction, enables us to automatically prove data properties as well as safety and liveness properties. In the following, we will give various examples of properties but related to specific privacy or accountability concerns.

We can prove dependent conflict properties as in [22, 23], for example:

```
SOMETIME EXISTS D:data NOT (PERMIT Kim.input[Hospital](D) AND
     DENY Kim.input[Hospital](D)).
```

But our conflict detection based on logical consistency is more general and safer.

Other properties are related to reachability. For instance [24] in the context of administrative RBAC and with specific algorithms, studies how an initial system

can reach a specific state. Defining the initial system and the expected final state this problem can be solved using property verification. Time, discrete or with dates, is useful in expressing the state change progression. A related example is the verification of purpose: does an agent performing an action is processing it according to the required purpose? There are several dedicated articles ([25,26]) on this subject but none with a property verification view. Since doing an action is constrained by its permission in the generated logical context, a simple schema for this property in AAL is

```
ALWAYS FORALL D:Data PERMIT KardioMon.processing(D, purpose).
```

Controlling data disclosure is the main concern of data privacy. In this case we are interested in situations where a piece of data reaches an authorized or non authorized agent. This can also be viewed as a reachability case but with emphasis on agent locations, with or without specific behavior for the agents. In [6] we show a related verification in the context of an healthcare system.

We also automatically prove several original accountability related properties. It is possible to automatically prove the natural criterion in Listing 17. We also prove three other properties which appear in Listing 18.

Listing 18 Accountability properties

```
// 1) validity of the sufficient audit condition
(ALWAYS AE => (ALWAYS SOMETIME AUDIT)) =>
     (CLAUSE(AUDIT, AE, UE, RE)
               => (ALWAYS ((NOT UE) => (SOMETIME RE))))
// 2) an equivalent formulation of CLAUSE(AUDIT, AE, UE, RE)
((ALWAYS (AE AND UE)) OR
(ALWAYS AE AND (UE UNTIL ((NOT UE) AND (ALWAYS AUDIT => RE)))))
// 3) decomposition of a complex contract
CLAUSE(AUDIT, AE1 AND AE2, UE1 AND UE2, RE1 AND RE2) =>
   CLAUSE(AUDIT, AE1, UE1, RE1) AND CLAUSE(AUDIT, AE2, UE2, RE2)
```

The first states that it exists a simple and sufficient audit condition to catch all the violations. The second is an equivalent formulation of the accountability clause interpretation of Listing 9. This clause says that either, in any state, the usages are correct or it is correct until a violation occurs then rectification happens in case of an audit. It is common that contracts are presented as a conjunction of policies. The third property expresses that such a complex accountability contract can be split in several sub-contracts and its verification implies the verification of its sub-contracts. The equivalence is possible for property 3) if the usage expressions are equivalent. These properties are rather intuitive or desirable but also improve efficiency in verification.

Other examples of properties and proofs are: *i)* a compliance criterion for the extended accountability template of Listing 13, however the compliance is covariant on all the parameters but contravariant on the audit expression, *ii)* the equivalent formulation for the extended template with the condition that `ALWAYS (VE => NOT UE)`, and *iii)* the decomposition property is also valid for the extended template in case of conjunction of policies.

A last usage of verification is relevant to the extended accountability template (from Listings 13 and 14) and its instantiation (Listing 7). Listing 19 shows the equivalence between both expressions and it can be proved in about two seconds. More precisely, with the set of expressions in Listing 14, we can prove that the example in Listing 7 is an instantiation of the extended template of Listing 13. This listing has three parts: *i)* the parameter instantiation and the conditions of the template, *ii)* the instantiated example coming from Listing 7, and *iii)* the template as in Listing 13. This expression is difficult to read but it is only for specialists of formal specifications which are in charge of defining the templates with their conditions and properties. The listing above illustrates the steps performed by the specialist to validate a new template. It starts from one or more concrete examples of accountability policy, it manually write the conditions and the template, next the specifier tries to automatically prove the expected instantiation.

Listing 19 Equivalence of the extended template instantiation

```
// i) instantiation from Listing 14
(ALWAYS (AUDIT <=> Auditor.audit())) AND
(ALWAYS (AE <=> (Kim.notify[Auditor]() => AUDIT))) AND
(ALWAYS (UE <=> FORALL D:Data (PERMIT Kim.send[Bob](D)
                    AND DENY Bob.send[Alice](D)))) AND
(ALWAYS (VE <=> EXISTS S:Data Bob.send[Alice](S))) AND
(ALWAYS (JE <=>
  (VE => ((@guilty(Bob) UNTIL Auditor.enforced[Bob]()) AND
          (@victim(Kim) UNTIL Auditor.enforced[Kim]())))))) AND
(ALWAYS (RE <=>
  (FORALL A:Agent (@guilty(A) => Auditor.punish[A]())) AND
  (FORALL A:Agent (@victim(A) => Auditor.compensate[A]()))))) AND
// template  conditions
(ALWAYS (VE => NOT UE)) AND
(ALWAYS ((JE AND VE) =>
  ((EXISTS A:Agent (@guilty(A) UNTIL enforced(A)))
   OR (EXISTS A:Agent (@victim(A) UNTIL enforced(A))))))
// -- end of conditions
=>
// ii) instantiated expression from Listing 7
ALWAYS (Kim.notify[Auditor]() => Auditor.audit()) AND
ALWAYS FORALL D:Data (Bob.send[Alice](D) =>
            (@guilty(Bob) UNTIL Auditor.enforced[Bob]())
             AND (@victim(Kim) UNTIL Auditor.enforced[Kim]())
AND (ALWAYS FORALL D:Data (PERMIT Kim.send[Bob](D)
                        AND DENY Bob.send[Alice](D))
    OR
       (FORALL D:Data (PERMIT Kim.send[Bob](D)
                       AND DENY Bob.send[Alice](D)))
     UNTIL ((EXISTS S:Data Bob.send[Alice](S)) AND
            (ALWAYS (Auditor.audit() =>
             (Auditor.judged() AND Auditor.punish[Bob]()
              AND Auditor.compensate[Alice]())))
// --
<=>
// iii) application of the extended template Listing 13
((ALWAYS (JE AND AE)) AND
((ALWAYS UE) | (UE UNTIL (VE AND (ALWAYS (AUDIT => RE))))))
```

Privacy officers are mainly concerned by choosing and filling the templates. In case of standard and simple regulation the privacy officer selects the right tem-

plate and its sub-expressions. Assistance could be improved by a dedicated graphic user interface. However, we claim that templates provide assistance specifically in case of complex regulations with several sub-contracts. Each sub-contracting has to formalize its proper part of the contract, maybe still using some templates. Afterwards the primary data processor should check the conditions of the template and then automatically apply it to the sub-contracts.

4 The AccLab Tool Support

In this section we sketch the main features of the AccLab tool support which represents a step in defining an end to end accountability framework from specification to implementation. AccLab is compound from a set of tools which are: The component editor, the AAL editor and its verifications, and the monitoring tools. The last release of AccLab is version 2.1 which was released on November 23, 2016 on GitHub (`https://github.com/hkff/AccLab`) under GPL3 license. The AccLab IDE is a web interface that provides a component diagram editor and tools to work with the AAL language. The back-end is written in Python3 and the front-end in JavaScript based on `dockspawn` (`http://www.dockspawn.com`) which is a web based dock layout engine released under MIT license. For verification purposes AccLab is interacting with the TSPASS tool. The implementation is still in progress we will give an overview of its main current features.

4.1 System Specification

We consider accountability by design that is going from specification to implementation of accountability. The starting point of the accountability process is a component diagram in the UML style which describes the application architecture. The method was described in [4] and allows the user to define in a graphic way the different agents and their required, provided and internal services. The diagram is enriched with textual annotations for services types and with accountability clauses associated to services and agents. To manage more easily the AAL language a dedicated editor has been implemented. This editor is directed by the syntax and highlights the language keywords. There are syntactic checks but also semantic controls for type checking and consistency of the declared services.

4.2 Verification Tools

A panel in the editor arranges a set of tools providing assistance in writing by the use of dedicated templates, for instance generating type declarations, accountability clauses or specific privacy expressions. This panel also contains few verification tools mainly the conflict checking with localization and the compliance checking. Figure 2 shows the graphical interface of AccLab, the AAL editor and its tool panel. Both checking tools use

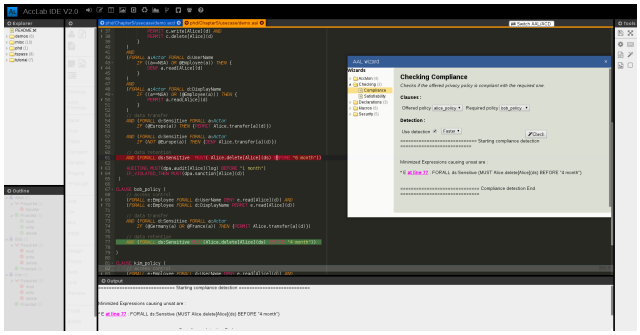

Fig. 2 AccLab screenshot

the connection with TSPASS and its satisfiability algorithm as explained in the previous section. This panel assists in generating macro calls which are useful in automating some complex tasks related to the translation to $\mathbb{FOTL}$ and the interaction with TSPASS.

4.3 Accountability in Action

One idea behind AccLab is to see accountability in action, one way to achieve that is to be able to run simulations. AccLab includes a simulation module that allows it to simulate agents in a system and to observe accountability in action. Each agent is wrapped in a reference monitor which acts as a proxy and intercepts all incoming and out-going messages of an agent. Reference monitors communicate with each other via a component that simulates the network. The agents are asynchronous and use asynchronous communications, and they exchange monitoring information when communicate with each others. The policy is monitored using an engine based on the rewriting technique [27, 28, 29, 30] and the monitors use distributed monitoring techniques [31]. [31] proposes local monitoring for distributed systems but limited to past formulae only. We reuse and extend this work focusing on future formulae and we use knowledge vectors (extending vector clocks) to update local monitors knowledge. Sub-monitors are

generated for each formula that describes non local actions in each agent. We extended the progression function to deal with knowledge vectors that are exchanged between agents during communications and containing the evaluation of sub-monitors.

4.4 Real Monitoring

We introduce the AccMon tool which implements the aforementioned principles, allowing for monitoring accountability policies in the context of real systems. The tool is build over the Django, an open-source web application framework in Python, relying on the model-view-controller pattern. AccMon allows to specify policies that are applicable to network traffic, web application code and other components via plug-ins. AccMon acts as a middleware in the Django framework. It intercepts and logs client's HTTP requests, server's requests processing and responses. On the web application side, the developer can configure the framework to intercept function/method calls and databases access. The properties to be monitored with the tool are defined in a variant of AAL with interpreted predicates. The tool receives log events via its RESTful API. AccMon can act as a daemon and interconnecting a variety of external tools.

In the following, we present a broad review of related works in Section 5. Experts in these areas will rather focus on Section 6, where we compare our work with similar initiatives in the field of computer science.

5 Formal Models for Accountability in Computer Science

Accountability is a complex and broad notion which has been discussed in several domains: economy, laws and regulations, ethics, privacy, education, public services, and much more. The term has recently evolved and gained much interest as analyzed in [9]. However, there is not a clear agreement on its characteristics [8] and how to make it computer understandable if possible. This is the main reason why we restrict our study to accountability in computer science. The notion of accountability crosscuts several domains of computer science: digital forensics, computer security, distributed systems in general (grid and cloud computing, the Internet and network applications) and natural language processing. The notion of accountability has been the subject of several surveys in computer science [32,33, 34,35]. We will here focus only on a set of papers related to computer science and accountability, focusing on formal models, specifications and verification. There are only few references in computer science which considers a general and interdisciplinary view of accountability, see [32,36,37]. Most of the papers, due to the complexity of the concept, only address some properties or specific mechanisms related to accountability. The preventive controls used by classical information technology security are not sufficient to achieve accountability. Accountability performs a posteriori control and it requires several mechanisms: information transparency, secure logging, checking misbehavior and responsibilities, then proceeding to penalties and compensation. There are already some proposals for frameworks integrating these aspects [37,21] and formal models or logic for accountability [38,39,40,41,42]. Recently Butin et al. advocate for strong accountability in [21]. The authors put forward strong accountability as a set of precise legal obligations supported by an effective software tool set. They demonstrate that the state of the art in term of technology is sufficient to ensure the notion of accountability by design. In the rest of this section we summarize some formal work around accountability since it is more closely related to our current work. There are also several interesting applications of the notion of accountability in concrete domains. Among them [43,44,45,46,47,48]. These approaches are dedicated to specific applications and use techniques that are not fully relevant to our abstract and formal context. We group the references of interest in three coarse parts: logical models, theories, and formal verification aspects.

5.1 Logical Models

These are logical models often with delegation. In [49] authors consider accountability transfer during right delegations. The basis of accountability is for a participant to prove a statement to a third party. This paper provides a rich logical model for communication protocols with authentication, a participant can formally prove some statements, is trusted on some statements or can exercise some rights. [50] focuses on distributing digital assets while preserving some privacy properties. The authors define a formal model to express usage policies and to enforce them. This is a logical model with agent creation, exchanging and redistributing data and assuming that an authority observes evidences of these actions. The associated proof system is based on predicate logic enriched with inference rules for communication, data and policy creation and delegation. The context of [38] is data privacy in a distributed system. The formal model allows to define accountability policies as extensions of First-Order-Logic ($\mathbb{FOL}$) with special constructions for data

ownership and right delegation but without negation. A terminating proof system for accountability is defined and implemented in the Twelf prover. The paper from Etalle and Winsborough [11] discusses the weaknesses of preventive security face to unanticipated situations occurring in collaborative environments. The authors argue that an approach based on deterrence is complementary and forms "a second line of defence". They define APPLE, a logical framework for a posteriori policy enforcement based on three critical components: logging, auditing, and accountability. The concept of sticky policy is used, it is a conjunction of few specific first-order predicates describing the owner and the permission to modify or distribute a document and to modify or refine a policy. An inference system can audit logs and proves that a user performed action in compliance with the sticky policy. A formal framework for privacy relying on accountability is proposed in [51]. The author considers privacy in modern pervasive systems and specifically the disclosure of privacy information. The author defines the SIMPL privacy language allowing features related to data disclosure. The semantics of agents is based on traces on which a notion of compliance is defined. The model of computation uses the notion of sticky policy and two global properties. These properties express that if a value is in the space of an agent, either the owner directly sent it to the agent, or the sticky policy enabled this agent to receive this value.

5.2 Theories

These are references focusing on abstract properties of audit, accountability and agent behaviours. In [39] the authors state that the accountability approach to security lacks general foundations for models and programming. They propose a theoretical operational model for accountability in a distributed system with definitions of honest agent, auditor, and responsiveness. This allows them to discuss the power of the auditor and the constraints placed on agents and on the communication infrastructure. The model is based on point to point communications providing integrity and authenticity guarantees. The behavior of agents is expressed via process algebra and discrete time. They use a game-based method, linear temporal logic and model-checking to check accountability properties. This model explores the trade offs between the honest principals, the communication network, the audit protocol and proposes five abstract properties about agent guilty blamed by the auditor. In [40] new definitions for accountability and verifiability are proposed and shown to be connected together. They provide two interpretations: A symbolic one in the Dolev-Yao style and a computational one with a cryptographic model. The authors demonstrate the applicability of their approach analyzing several cryptographic protocols. The authors of [41] claim to provide a more general and more widely applicable definition of accountability. They explain that existing approaches have been mainly preventive which is inadequate since it is generally impossible to, a priori, differentiate an honest user from an attacker. A posteriori or corrective approach is being more suitable to accountability. They provide a formal model of accountability based on event traces and utility functions taking into account anonymous agents, automatic and mediated enforcement of accountability.

5.3 Formal Verification

The work discussed in this section is related to concrete approaches targeting verification means for specific domains. Note that [38] suggests a limited tool support. AIR [20] (Accountability in RDF) is a rule-based language for the semantic Web and supports rule nesting, and explanation of inferences. AIR supports a non-monotonic negation and rules ordering counts. The semantics is based on defining the translation of an AIR-program to a semantically equivalent stratified Logic Program. It employs a RETE based forward-chaining approach to compute the AIR-closure and allows closed-world reasoning. The language does allow neither permissions nor features for rectifications. The accountability views of AIR is only covering explanations and justifications needed for the audit task. [52] provides a formal service contract for accountable SaaS services. The authors, after analyzing some business contract languages, identify few requirements, a major one for them is language decidability. They propose a formal model, called OWL-SC, and a representation of the contracts based on ontology and mixes two languages OWL-DL and SWLR. They also present a translation of these contracts into the formalism of colored Petri nets. This allows to check properties and to reason on the contracts with CPN-tools. [53] proposes a formal model for service composition with accountability, which is called Accountable Cloud Service (ACS). The language enables the expression of deontic constructions: obligatory, permissible, and prohibited actions as well as rules for remediation. The semantics is based on dynamic logic extended with deontic and accountability constructions but the authors claim to remove any paradox using the Dynamic Logic. ACS provides a notation for modeling service collaboration based on BPMN 2.0 and proposes an Obligation Flow

Diagram as a method for conflict resolution and verification. ACS allows the specification of accountability contracts in a machine understandable style but does not yet pave the way for a tool support.

6 AAL and AccLab Discussion

We introduce our general approach for accountability in [5] with the idea to focus on enforcement by synthesizing an XACML [54] extension. [4] presents a component-based approach to specify accountability in a system and a model-checking based verification approach. We switch to a more abstract approach using the First-Order linear Temporal Logic and the TSPASS prover in [18,6]. In [7] we demonstrate that AAL is suitable for several kinds of security and privacy concerns. We also analyze the problem of conflict detection in policy sets, exhibit weaknesses of most of the current proposals and justify the choice to rely on logical consistency. A real accountability policy example is discussed in [55] as well as lessons learnt from its formal specification with our language. The current paper summarizes our previous work but adds a bundle of related work and comparisons. We also discuss more precisely how to write accountability expressions and some limits of the current formalism. We provide new unique features: the generalized authorizations and the accountability templates. We also expose the verification principles, behind our tool like conflict detection, and compliance. The AccLab tool description and its monitoring facilities were also never published.

6.1 Comparison with Related Work

[53] proposes a formal model for service composition with accountability, which is called Accountable Cloud Service. The logic is not a classic one and is mixed with BPMN notations, the drawback is that neither a decision algorithm nor a tool support are described. Another close work is [38] which defines a policy language based on $\mathbb{FOL}$ with policy disclosure allowing delegation responsibility. The main difference is that we consider policies already defined and assigned, without a native delegation mechanism. But note that the same compliance relation is valid in both contexts. It also proposes a concrete tool support with the Twelf proof checker but AAL adds linear temporal features and AccLab automated proofs. [11] is another formal model for accountability, AAL is more abstract because we do not assume specific component like trust management and dedicated predicates for refinement or transfer. There are only two previously existing tools [52,20]. On one hand, the language proposed by [52] is limited to ontology, without specific feature for audit or rectification and targeted to monitoring. On the other hand [20] focuses on explanations of inferences. Our approach is unique on two sides: the language and its tool support. Regarding AAL its unrivalled features are its $\mathbb{FOTL}$ foundation with an accountability clause, general authorizations and templates. AccLab has the following noteworthy characteristics: an expressive language covering most of the security and accountability needs as well as formal compliance, conflict checking and monitoring support.

Aside the previous accountability articles there is other related work which crosscuts our approach. Accountability like security is expressed by policies thus many works done in the policy domain are related to our. Our work, especially the usage expression part is related to privacy and security languages like: [56,57, 58,59]. In addition to specific accountability features (audit and rectification) our work has the following characteristics. As demonstrated in [7] the language provides negation, even for obligation, access control and privacy concerns thus covering many needs. We rely on the $\mathbb{FOTL}$ framework with the monodic constraint, this is related to [56], but without the burden of fix points. While [59] present a $\mathbb{FOL}$ approach for privacy and accountability enforcement but without explicit linear temporal operators.

Regarding policy verification we think that [60,22, 61,62] are the most related, all are limited to usual security policies. [60] is only access control in the $\mathbb{FOL}$ context, [22] focuses on location properties without tool support, and [61] uses $\mathbb{FOL}$ with event calculus and a verification prototype restricted to finite domains. While [62] defines a rich policy language in Fusion logic and uses model-checking.

Enforcement of accountability has been done in a specific and concrete way in [63] with an extension of XACML. However, A-PPL (An Accountability Policy Language) is not a flexible approach since it operates with few dedicated constructions. It also inherits various limitations from its XACML ancestor. A more abstract vision of enforcement is proposed by monitoring or runtime verification [64]. Our monitoring approach is rather close to the following papers [31,27,30]. In fact we mix the use of rewriting temporal expressions, the temporal and first-order approach, localization and monitoring distributed systems.

6.2 Open Questions and Limits

In the previous sections, we have seen several ways to express accountability with AAL. Regarding the three

dimensions of accountability from [8], *i)* we have *information* as formal contract which can be checked for compliance, *ii) justification* is provided by querying the formal contract using property verification, and *iii) punishment* is made explicit in the contract and enforced by monitoring.

It is possible to embed models like [49,50,38,11,51] as they are based on predicate calculus, first-order inference rules and linear traces. The model in [53] uses a dynamic deontic logic based on $\mathbb{FOL}$ and a discrete temporal operator. The definition of the accountability clause is rather limited and specific, because it considers accountability related to one atomic action. In AAL we keep a classic logic with its tool support. We have orthogonal notions of permission, and action but not the exact obligation of deontic logic. Paradoxes of deontic logic seem difficult, if possible, to remove [65]. Another set of related work is focusing on properties for audit and accountability [39,41,40]. These are more challenging or impossible to fully encode without strongly restricting expressiveness since they are based on utility functions or probabilistic approaches.

We aim at integrating other notions of accountability, those defined outside computer science, as in [8,9]. However, one first limitation is that these are complex notions and there is no clear agreement on precise and operational definitions of accountability characteristics. The second point is that some characteristics (for instance, moral dilemma or force majeure) are specific of human behaviors and cannot be modelled or are out of the scope of automation. Many works consider that true accountability cannot be completely automatic and humans (an auditor, a judge, etc) should be involved in the process. This is still according to [8]: "Given that the notion of accountability is not built on the illusion that power is subject to full control ...". A fully automated solution would be equivalent to a preventive security solution, all violation cases and countermeasures are known and decided in advance. In our approach human behavior is defined by actions connected to a virtual agent (as in [51]) under the control of the real human, thus it will be transparent here.

Other difficult notions are linked to deontic concepts, like the contrary to duty. Finally, there are some concepts like complicity, valid excuse, group accountability, grades of accountability which are totally or partly machine understandable. The main barrier for us is to obtain an agreement or at least sufficiently precise definitions for these concepts in order to model them. There are also expressiveness and decidability issues in accountability policies. However many progresses have been done recently in satisfiability modulo theories and we could expect new results in automated verification of $\mathbb{FOTL}$.

7 Conclusion and Perspectives

Accountability is a complex concept that becomes more important in the digital society, as an effect of the raising privacy concerns in ubiquitous systems. We advocate a privacy by design approach, addressing accountability requirements starting from specification towards their implementation in software systems. We provide a flexible and expressive domain agnostic language, where one can handle distinct usages and definitions for the term accountability. Accountability interpretation and its operational management are slightly varying in the related work, even from a formal perspective. This lead us naturally to the development of the accountability laboratory, supported by its verification principles and concrete tools for policy verification and monitoring. The existing related work is rather limited to theoretical models without tool support. Tools support for some dedicated applications and domains are proposed in [20, 52]. However, a wider perspective with concrete language and enforcement tools was lacking. In this paper we demonstrate that this objective is perfectly attainable and our proposal is aligned with the three Schedler dimensions [8]. We provide the AAL language with semantics based on $\mathbb{FOTL}$, we demonstrate its expressiveness and flexibility concerning accountability policies. Apart from its expressiveness AAL provides a notion of general authorizations and a convenient notion of templates to assisting in writing policies. We study verification means for internal and specific properties and we propose conflict detection, and policy compliance. Finally, these ideas have been implemented in our AccLab tool support and it provides accountability writing, consistency checking, compliance verification, and runtime monitoring.

In the near future, we intend to improve the language and its tool support. In fact one challenge is the integration of more computation primitives in $\mathbb{FOTL}$. We also expect to relax the monodic constraint. For instance, `(always forall X,Y pred(X, Y)) => forall X,Y always pred(X, Y))` is a valid property. While the conclusion of the right-hand side is not monodic, it can be proved with the left-hand side. With respect to performance, in [6] we defined some heuristics to reduce execution time and we expect to rationalize them. One important topic, which is actually not discussed in this paper, is the link between the global view promoted by AAL and the local view needed for distributed agents. Still, there are some decidability issues but results exist with the session type theory [66].

References

1. Regulation, E. U. REGULATION (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016. `http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf`, 2016. accessed by 22/05/2017.
2. 104th Congress Public Law 191. Health Insurance Portability and Accountability Act of 1996. `https://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm`, 1996. accessed by 22/05/2017.
3. 107th Congress Public Law 204. Corporate and Auditing Accountability and Responsibility Act. `https://www.gpo.gov/fdsys/pkg/PLAW-107publ204/html/PLAW-107publ204.htm`, 2016. accessed by 22/05/2017.
4. Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed Sellami. Accountability for Abstract Component Design. In *EUROMICRO DSD/SEAA 2014*, pages 213–220, Verona, Italie, August 2014.
5. Walid Benghabrit, Hervé Grall, Jean-Claude Royer, Mohamed Sellami, Melek Önen, Anderson Santana De Oliveira, and Karin Bernsmed. volume 512 of *Communications in Computer and Information Science*, chapter From Regulatory Obligations to Enforceable Accountability Policies in the Cloud, pages 134–150. 2015.
6. Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed Sellami. Abstract accountability language: Translation, compliance and application. In *APSEC*, New Delhi, India, 2015. IEEE Computer Society.
7. Jean-Claude Royer and Anderson Santana De Oliveira. Aal and static conflict detection in policy. In *CANS, 15th International Conference on Cryptology and Network Security*, LNCS, pages 367–382. Springer, November 2016.
8. Andreas Schedler. *Self-Restraining State: Power and Accountability in New Democracies*, chapter Conceptualiazing Accountability, pages 13–28. Lynne Reiner, 1999.
9. Richard Mulgan. 'accountability': An ever-expanding concept? *Public Administration*, 78(3):555–573, 2000.
10. Walid Benghabrit and Hervé Grall and Jean-Claude Royer. Accountability Laboratory. `http://www.emn.fr/z-info/acclab/`, 2016. accessed by 22/05/2017.
11. Sandro Etalle and William H. Winsborough. A posteriori compliance control. In Volkmar Lotz and Bhavani M. Thuraisingham, editors, *SACMAT 2007*, pages 11–20. ACM, 2007.
12. Michel Ludwig and Ullrich Hustadt. Implementing a fair monodic temporal logic prover. *AI Commun*, 23(2-3):69–96, 2010.
13. Michel Ludwig. Tspass, 2010. https://lat.inf.tu-dresden.de/~michel/software/tspass/.
14. Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable fragment of first-order temporal logics. *Ann. Pure Appl. Logic*, 106(1-3):85–134, 2000.
15. Anatoli Degtyarev, Michael Fisher, and Boris Konev. Monodic temporal resolution. *ACM Transactions on Computational Logic*, 7(1):108–150, January 2006.
16. Carmen Fernández-Gago, Ullrich Hustadt, Clare Dixon, Michael Fisher, and Boris Konev. First-order temporal verification in practice. *Journal of Automated Reasoning*, 34(3):295–321, 2006.
17. Viktor Schuppan and Luthfi Darmawan. Evaluating LTL satisfiability solvers. In Tevfik Bultan and Pao-Ann Hsiung, editors, *ATVA*, volume 6996 of *LNCS*, pages 397–413. Springer, 2011.
18. Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed Sellami. Checking Accountability with a Prover. In *COMPSAC*, pages 83–88, Taichung, China, 2015.
19. Viktor Schuppan. Towards a notion of unsatisfiable and unrealizable cores for LTL. *Science of Computer Programming*, 77(7–8):908–939, July 2012.
20. Ankesh Khandelwal, Jie Bao, Lalana Kagal, Ian Jacobi, Li Ding, and James A. Hendler. Analyzing the AIR language: A semantic web (production) rule language. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR*, volume 6333 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2010.
21. Denis Butin, Marcos Chicote, and Daniel Le Métayer. Strong Accountability: Beyond Vague Promises. In *Reloading Data Protection: Multidisciplinary Insights and Contemporary Challenges*, pages 343–369. Springer, 2014.
22. Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. Logic-based conflict detection for distributed policies. *Fundamantae Informatica*, 89(4):511–538, 2008.
23. Bei Wu, Xing yuan Chen, Yong fui Zhang, and Xiang dong Dai. An extensible intra access control policy conflict detection algorithm. In *Computational Intelligence and Security*, pages 483–488. IEEE Computer Society, 2009.
24. Scott D. Stoller, Ping Yang, C. R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In *Conference on Computer and Communications Security*, pages 445–455, 2007.
25. Lili Sun, Hua Wang, Xiaohui Tao, Yanchun Zhang, and Jing Yang. Privacy preserving access control policy and algorithms for conflicting problems. In *TrustCom*, pages 250–257. IEEE Computer Society, 2011.
26. Michael Carl Tschantz, Anupam Datta, and Jeannette M. Wing. Purpose restrictions on information use. volume 8134 of *ESORICS 2013*, pages 610–627. Springer, 2013.
27. Grigore Rosu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng*, 12(2):151–197, 2005.
28. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.
29. Torben Scheffel and Malte Schmitz. Three-valued asynchronous distributed runtime verification. In *Formal Methods and Models for Codesign*, pages 52–61, Oct 2014.
30. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. The ins and outs of first-order runtime verification. *Formal Methods in System Design*, 46(3):286–316, 2015.
31. Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *International Conference on Software Engineering*, pages 418–427, May 2004.
32. Daniel J. Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James Hendler, and Gerald Jay Sussman. Information accountability. *Commun. ACM*, 51(6):82–87, June 2008.
33. Kwei-Jay Lin, Joe Zou, and Yan Wang. Accountability computing for e-society. In *24th Advanced Information Networking and Applications Conference (AINA)*, pages 34–41. Ieee, 2010.
34. Yang Xiao Zhifeng Xiao, Nandhakumar Kathiresshan. A survey of accountability in computer networks and distributed systems. *Security and Communication Networks*, 2012.
35. Daniel Guagnin, Leon Hempel, Carla Ilten, Inga Kroener, Daniel Neyland, and Hector Postigo, editors. *Managing Privacy through Accountability*. Palgrave Macmillan, 2012.

36. Daniel Le Métayer. Formal methods as a link between software code and legal rules. *Software Engineering and Formal Methods*, pages 3–18, 2011.
37. Siani Pearson and Nick Wainwright. An interdisciplinary approach to accountability for future internet service provision. *International Journal of Trust Management in Computing and Communications*, 1(1):52–72, 2013.
38. Jan Cederquist, Roberto Corin, Marnix Dekker, Sandro Etalle, and Jerry Den Hartog. An audit logic for accountability. In *POLICY'05*, pages 34–43. IEEE, 2005.
39. Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. Towards a theory of accountability and audit. ESORICS'09, pages 152–167, Berlin, Heidelberg, 2009. Springer-Verlag.
40. Ralf Kusters, Tomasz Truderung, and Andreas Vogt. Accountability: definition and relationship to verifiability. pages 526–535. ACM, 2010.
41. Joan Feigenbaum, Aaron D. Jaggard, and Rebecca N. Wright. Towards a formal model of accountability. In Sean Peisert, Richard Ford, Carrie Gates, and Cormac Herley, editors, *New Security Paradigms Workshop*, pages 45–56. ACM, 2011.
42. Joan Feigenbaum, Aaron D. Jaggard, and Rebecca N. Wright. Open vs. closed systems for accountability. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, pages 4:1–4:11, New York, NY, USA, 2014. ACM.
43. Brent N. Chun and Andy Bavier. Decentralized trust management and accountability in federated systems. *Hawaii International Conference on System Sciences*, 9:90279a, 2004.
44. Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 119–134, 2010.
45. Smitha Sundareswaran, Anna Cinzia Squicciarini, and Dan Lin. Ensuring distributed accountability for data sharing in the cloud. *IEEE Trans. Dependable Sec. Comput*, 9(4):556–568, 2012.
46. Ryan K. L. Ko, Bu-Sung Lee, and Siani Pearson. Towards achieving accountability, auditability and trust in cloud computing. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *Advances in Computing and Communications*, volume 193, pages 432–444. Springer, 2011.
47. Siani Pearson. Accountability in cloud service provision ecosystems. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems - 19th Nordic Conference*, volume 8788 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2014.
48. Anupam Datta. *Privacy through Accountability: A Computer Science Perspective*, pages 43–49. Springer International Publishing, Cham, 2014.
49. Bruno Crispo and Giancarlo Ruffo. Reasoning about Accountability within Delegation. In *International Conference on Information and Communications Security*, pages 251–260. Springer-Verlag, 2001.
50. Roberto Corin, Sandro Etalle, Jerry den Hartog, Gabriele Lenzini, and I. Staicu. *A Logic for Auditing Accountability in Decentralized Systems*, pages 187–201. Springer US, Boston, MA, 2005.
51. Daniel Le Métayer. *A Formal Privacy Management Framework*, pages 162–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
52. Joe Zou, Yan Wang, and Kwei-Jay Lin. A formal service contract model for accountable saaS and cloud services. In *International Conference on Services Computing*, pages 73–80. IEEE Computer Society, 2010.
53. Jun Zou, Yan Wang, and Mehmet A. Orgun. Modeling accountable cloud services. In *International Conference on Web Services*, pages 353–360. IEEE, 2014.
54. OASIS Standard. eXtensible Access Control Markup Language (XACML) Version 3.0. 22 January 2013. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`, 2013. accessed by 22/05/2017.
55. Walid Benghabrit, Jean-Claude Royer, and Anderson Santana De Oliveira. Towards the specification of natural language accountability policies with acclab: The laptop policy use case. volume 2051, October 2017. http://ceur-ws.org/.
56. Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In *ACM Workshop on Privacy in the Electronic Society*, pages 73–82, 2010.
57. Moritz Y. Becker, Alexander Malkis, and Laurent Bussard. A practical generic privacy language. volume 6503 of *ICISS 2010*, pages 125–139. Springer, 2010.
58. Guillaume Piolle and Yves Demazeau. Representing privacy regulations with deontico-temporal operators. *Web Intelligence and Agent Systems*, 9(3):209–226, 2011.
59. Anupam Datta, Jeremiah Blocki, Nicolas Christin, Henry DeYoung, Deepak Garg 0001, Limin Jia, Dilsun Kirli Kaynar, and Arunesh Sinha. Understanding and protecting privacy: Formal semantics and principled audit mechanisms. In Sushil Jajodia and Chandan Mazumdar, editors, *Information Systems Security*, volume 7093 of *LNCS*, pages 1–27. Springer, 2011.
60. Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. *ACM Transactions on Information and System Security*, 11(4):1–41, July 2008.
61. Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil C. Lupu, and Arosha K. Bandara. Expressive policy analysis with enhanced system dynamicity. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security*, pages 239–250. ACM, 2009.
62. Antonio Cau, Helge Janicke, and Ben C. Moszkowski. Verification and enforcement of access control policies. *Formal Methods in System Design*, 43(3):450–492, 2013.
63. Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana de Oliveira, and Jakub Sendor. A-PPL: an accountability policy language. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 319–326, 2014.
64. Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions, 2010.
65. Albert J. J. Anglberger. Dynamic deontic logic and its paradoxes. *Studia Logica*, 89(3):427–435, 2008.
66. Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, July 2016.